Energy-Efficiency Tactics in Software Architecture and Implementation

Agile7 Whitepaper

Document Version: v1.1

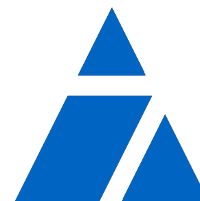
Date: October 23, 2024

Author: Max Meinhardt

Agile7

maxm@agile7.com

www.agile7.com



Executive Summary

In recent decades, software development has prioritized modularity and economies of scale, leading to rapid and cost-effective implementation but compromising energy efficiency. However, as energy costs rise and the world embraces renewable energy, the importance of software energy-efficiency processes, tools, and best practices has grown significantly. To address this, it is crucial to identify software architecture and implementation tactics that enhance energy efficiency while minimizing negative impacts on application speed. Incorporating these tactics during software architecture and automating them with simple and transparent DevOps tools will be essential for consistently creating energy-efficient software.

Agile7 Roadmap

Agile7 is implementing the following four phases in order to help make software more energy efficient. This whitepaper falls under Phase 2 shown below.

Phase 1: Identify the Business Landscape
- Identify risks in the ICT industry macro environment that would be caused by a continuous expansion of carbon emission regulations and energy-efficiency requirements. Find strategies to mitigate those risks and then identify their contingencies.
- Given this macro environment, identify the point at which the adoption of software energy-efficiency services reaching critical mass can be identified.

Phase 2: Identify the Tactics
➡ - Identify, group, and categorize the architectural tactics that can be used to improve software energy-efficiency. Give implementation suggestions in some tactics.
- Document hardware and software-specific energy-optimization tactics.

Phase 3: Create Processes
- Create processes for tactics from the previous step and place them into a hierarchical structure that is grouped into one or multiple orchestration categories.
- Identify process and orchestration gaps in order to identify needed tools for the next phase.

Phase 4: Create Tools
- Identify existing tools and create new tools that are used to fulfill process requirements from the previous phase.

Table of Contents

Executive Summary....... 2
Agile7 Roadmap....... 3
Introduction....... 6
Software Energy-Efficiency Tactics....... 7
1) Resource Allocation....... 8
1.1) Scaling (cloud-specific)....... 8
1.2) Scheduling....... 8
1.3) Service Brokering (cloud-specific)....... 8
2) Resource Adaptation....... 9
2.1) Reduce Overhead....... 9
2.1.1) Make Software Cloud-Native (cloud-specific)....... 9
2.1.2) Adopt Use-Case-Driven Design....... 9
2.2) Service Adaptation....... 10
2.3) Increase Efficiency....... 10
2.3.1) Make Resources Static....... 10
2.3.2) Apply Edge Computing....... 10
2.3.3) Apply More Granular Scaling....... 11
2.3.4) Choose Optimal Deployment Paradigm (cloud-specific)....... 12
2.3.5) Caching....... 12
2.3.5.1) Web Caching....... 12
2.3.5.1.1) HTTP Cache Headers....... 12
2.3.5.1.2) Web Application-Level Caching....... 13
2.3.5.1.3) Edge Side Includes....... 13
2.3.5.2) Cloud Caching (cloud-specific)....... 13
2.3.6) Optimize Search and Query Strategies....... 14
2.3.6.1) SQL Optimization....... 14
2.3.6.2) NoSQL Optimization....... 14
2.3.7) Compress Infrequently Accessed Data....... 14
2.3.7.1) Application Data Compression....... 14
2.3.7.2) Cloud Database Compression (cloud-specific)....... 15
2.3.8) Optimize Code....... 15
2.3.8.1) Computational Efficiency....... 15
2.3.8.1.1) Algorithm Design....... 15
2.3.8.1.1.1) Energy-Efficient-Algorithm Design Process....... 16
2.3.8.1.2) Simplify Boolean-Logic Expressions....... 17
2.3.8.2) Low-Level or Intermediate Code Optimization....... 17
2.3.8.2.1) Use the Most Optimal Computer Language....... 17
2.3.8.3) Parallelism....... 19
2.3.8.3.1) DVFS....... 19
2.3.8.3.1.1) Energy Profiling for DVFS....... 19
2.3.8.4) Data and Communications Efficiency....... 20

3) Resource Monitoring....21
3.1) Application Energy Monitoring....21
3.1.1) Cloud Energy Monitoring (cloud-specific)....21
3.1.2) Performance Tools & Energy To Performance Correlation....22
3.2) OS Power Monitoring....22
3.3) Hardware Energy Monitoring....22
3.3.1) Measuring Application Energy using a Hardware Power Monitor....23
3.3.2) Hardware Power Monitoring Devices....24
Conclusion....25
Learn How Improving Software Energy-Efficiency Can Help You....25
About Agile7....25
Document Revision History....26
Author Biography....26
References....27

Introduction

Multiple sources [6][9] demonstrate that the tactics described in this whitepaper can reduce software energy consumption by 5% to over 50% while minimizing performance loss. However, several factors can influence these outcomes. For instance, consider the comparison between a polled event handler and an interrupt-driven event handler, both delivering the same performance. Yet, their energy efficiency can significantly vary depending on the polling frequency and event occurrence. During software architecture and development, it is essential to analyze such issues and others to maximize software energy efficiency and maintain awareness of it.

The need for this awareness will continue to intensify as energy prices rise and carbon emission regulations become stricter, particularly in industries responsible for a significant carbon footprint. Notably, the ICT sector is increasingly contributing to this footprint. If its growth rate matches that of 2007 to 2020, global greenhouse gas emissions from the ICT sector will increase from 3.0–3.6% in 2020 to 14% by 2040, as per Vos et al. [41]. This substantial shift in global macroeconomic policies will prompt the integration of energy-efficiency best practices, tools, and process automation into the software industry, akin to what has been achieved in the cybersecurity industry.

This white paper explains these best practices. Specifically, the software architecture tactics used to decrease the energy use of Web and IoT applications in native and cloud-native environments. For each of these tactics, suggestions are shown for the architect or developer to implement.

Software Energy-Efficiency Tactics

Figure 1 provides a summary of tactics for architecting energy-efficient software, partially derived from the research paper entitled "Architectural Tactics to Optimize Software for Energy Efficiency in the Public Cloud" (Vos et al. [41]).

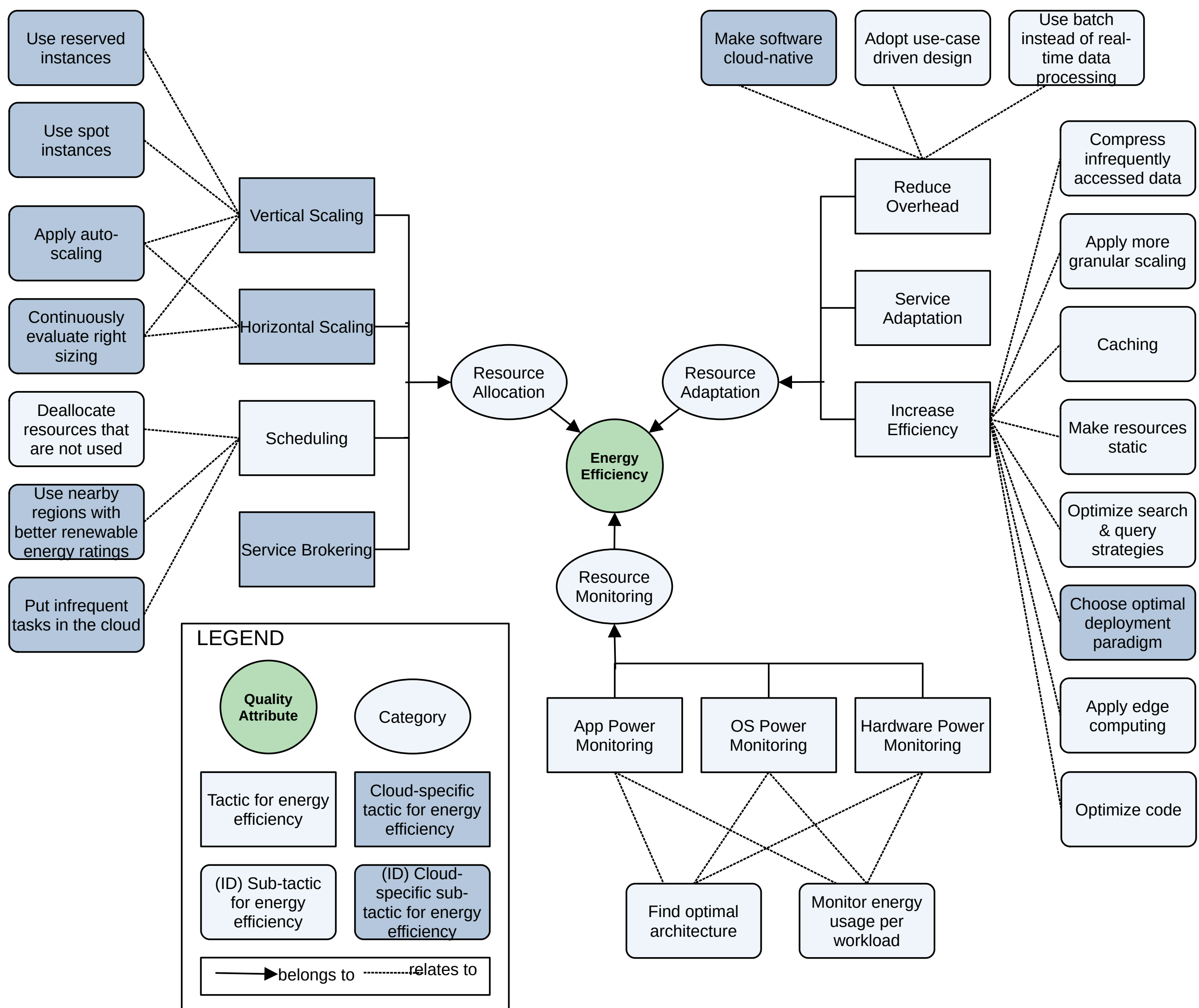


Figure 1: Software Energy-Efficiency Tactics

1) Resource Allocation

Software energy-efficiency resource allocation tactics encompass strategies that involve allocating and de-allocating resources to distribute processing time effectively and minimize overhead.

1.1) Scaling (cloud-specific)

The impact of energy usage on resource scaling (auto scaling) depends on how the cloud provider implements the underlying resource orchestration. In certain cases, the implementation may involve pre-allocation of resource bandwidth, such as when using reserved instances. This means that specific attributes (such as instance type or region) have their on-demand resource bandwidth pre-purchased within the cloud, resulting in discounted pricing. On the other hand, in situations like AWS spot instances, unused on-demand cloud storage capacity can be purchased at a discounted rate.

While both examples demonstrate a direct correlation between energy utilization and cost savings, there are some considerations to keep in mind. In the former case, if the sizing is not periodically evaluated, it can lead to increased overhead. The latter option is cost-effective only if you have flexibility in determining when your application runs since its pricing is influenced by the existing demand for the service, which can vary throughout the day.

When optimizing energy efficiency, it is crucial to adhere to the best practices of the orchestration environment when configuring both vertical and horizontal auto-scaling functionality.

1.2) Scheduling

To optimize power consumption and minimize the impact on energy usage, it is crucial to optimize the scheduled timing of process-intensive activities and resource deallocation based on traffic patterns. A cloud-based strategy example involves utilizing the AWS AMS Resource Scheduler [17] to automatically schedule the start and stop of Amazon EC2 instances, as well as scaling of Auto-scaling groups [48], prior to expected changes in traffic patterns.

For OS-specific scheduling, it is advisable to configure tools like cron to execute necessary tasks only during intervals when its system has the lowest load. By scheduling these tasks during periods of the day when the application experiences the lowest throughput, the likelihood of energy-inefficient events, such as network collisions, is reduced. This approach helps to mitigate the potential impact on power consumption.

1.3) Service Brokering (cloud-specific)

According to Gartner [12], a cloud service broker (CSB) is an “IT role and business model in which a company or entity adds value to one or more cloud services (public or private) on behalf of consumers of those services. The CSB performs three primary roles: aggregation, integration, and customization

brokerage.” There are three main types of CSBs: cloud aggregator, cloud integrator, and cloud customizer.

A cloud aggregator integrates multiple service catalogs, while a cloud integrator automates workflows across hybrid environments through a single orchestration. On the other hand, a cloud customizer modifies existing cloud services and adds new ones based on customer requirements.

Implementing a CSB can lead to cost and energy savings by reducing the allocation of unused cloud services that would otherwise go to waste if purchasing packaged feature sets that include unnecessary services. By leveraging a CSB, companies can optimize resource utilization and avoid unnecessary energy consumption.

2) Resource Adaptation

Software energy-efficiency resource adaptation strategies focus on optimizing hardware and software resource architectures to improve software efficiency [41]

2.1) Reduce Overhead

Overhead refers to excess software functionality and resources that are unused or can be reduced in size and scope without compromising performance. It often arises from factors such as over-engineering, inadequate integration leading to the presence of “zombie” code, and other similar issues.

2.1.1) Make Software Cloud-Native (cloud-specific)

Cloud-native software applications comprise microservices that are often packaged into containers, seamlessly integrating with cloud environments. These microservices collaboratively form an application, with each having the ability to scale independently and undergo continuous improvement and iteration through orchestration and automation processes. The inherent flexibility of microservices contributes to the agility and continuous enhancement of cloud-native applications, enabling them to adapt and leverage cloud resources for optimized performance, cost-efficiency, and energy efficiency.

Furthermore, this architecture enables hosting shared microservices across multiple applications, such as both web and mobile native apps, facilitated by service brokering. It also facilitates the separation of application functionality, allowing different development teams to be assigned to specific tasks.

A common use-case for transforming an application into a cloud-native one involves migrating a web application from a monolithic architecture (e.g., one that relies on a web server) to a containerized microservices architecture. This entails refactoring the software into multiple microservices. Despite the potential overhead resulting from the additional software generated during this process, the cloud provider’s energy-saving features outweigh this overhead, resulting in overall energy efficiency gains.

2.1.2) Adopt Use-Case-Driven Design

The use-case-driven design approach, also known as the “one-client approach” to software architecture, typically entails lower overhead compared to a domain-driven design, which emphasizes scalability. However, this architecture demands more careful planning and a deep understanding of the product’s long-term business logic compared to the use-case design’s focus on specific problem-solving aspects that are often easier to quantify during the early stages of software design.

During the software implementation phase, the use-case design approach prompts developers to address technical details sooner rather than later, enabling faster coding. However, this expedited approach may sacrifice scalability and introduce maintenance challenges down the line. To mitigate these concerns, it is recommended to combine multiple design methodologies, striking a balance that minimizes unnecessary layers of abstraction and redundant functionality while achieving the ultimate goal of efficient and maintainable software architecture.

2.2) Service Adaptation

In the context of service adaptation, Vos et al. [1] emphasizes the importance of selecting services based on energy-related information. For instance, a cloud service broker (CSB) can distinguish itself by incorporating energy-related data into its service APIs, offering comprehensive application-energy data collection, and providing graphical visualization capabilities in its user interface. Ideally, leveraging the energy data from these APIs enables applications to develop a more precise energy model, enhancing the accuracy of static code energy analyzers and runtime energy optimization middleware (section 2.3.8.3.1), such as DVFS manipulation.

2.3) Increase Efficiency

These are tactics for utilizing resources efficiently in order to optimize energy-efficiency.

2.3.1) Make Resources Static

Maximizing the utilization of static resources leads to improved software performance and long-term energy efficiency by minimizing real-time processing requirements. For instance, a static website can load up to 10 times faster (Deszkewicz [18]) compared to a dynamic website generated with a content management system (CMS). When considering the cumulative effect over time, the energy savings can be significant.

2.3.2) Apply Edge Computing

To mitigate the energy consumption associated with data transmission over the Internet, it is beneficial to bring remote services closer to the devices through the use of edge computing technology.

In the context of web services, a content delivery network (CDN) serves as an early form of edge computing, focusing primarily on caching data closer to the end user. Modern edge computing goes beyond caching and brings cloud services and storage closer as well. This approach offers the advantages of reduced energy usage and network latency, particularly valuable for the increasing number of IoT devices projected to be deployed in the future. Figure 2 below shows a generic model for an IoT edge network.

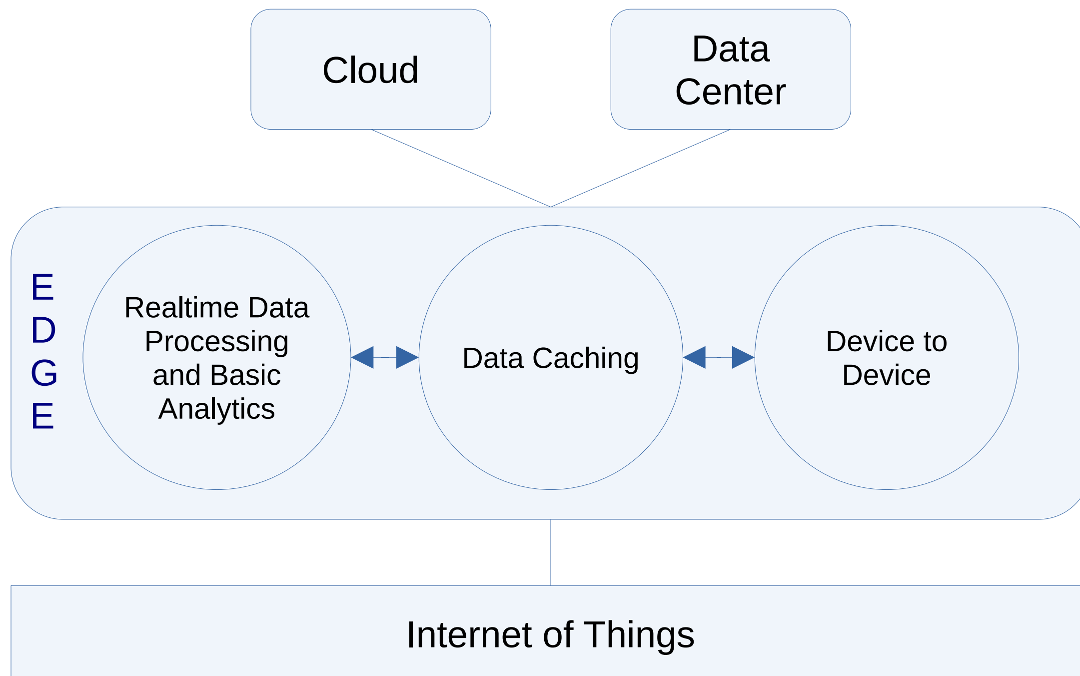


Figure 2: Generic IoT Edge Network

2.3.2.1) Network Traffic Asymmetry

As per Vailshery [13], the number of IoT devices worldwide is projected to more than double from 2022 to 2030, and many of these devices, such as cameras and sensor-based devices, will require more upstream data transmission than downstream.

Currently, most internet traffic is transmitted downstream to media-streaming devices and PCs. However, failure to consider this reversal of traffic direction when selecting network interconnect devices for an IoT network can lead to network congestion issues and a significant increase in power consumption.

2.3.3) Apply More Granular Scaling

Granular scaling is essentially the splitting of a workload into more manageable components so that its resource utilization can be optimized. Here are a few examples:

- Converting a single-threaded firmware architecture into a multi-threaded one (section 2.3.8.3), allowing for thread allocation across multiple CPU cores with different frequencies (p-states, section 2.3.8.3.1) and power requirements. This is dependent on the configuration of the CPU power governor.
- Splitting services and functionalities during the code refactoring process when transitioning applications to a cloud-native (section 2.1.1) architecture.
- In cloud auto-scaling, fine-grained scaling can be achieved by customizing the scaling based on specific types, such as APIs, where certain APIs may experience higher traffic than others.

2.3.4) Choose Optimal Deployment Paradigm (cloud-specific)

The choice of deployment paradigm among containers, virtual machines (VMs), and serverless architectures should be based on the characteristics of the cloud workload to optimize energy utilization, as stated by Vos et al. [41]. For instance, VMs are most effective for stable and predictable workloads, while serverless architectures are suitable for bursty workloads.

2.3.5) Caching

Caching is a highly effective approach to enhance energy efficiency, and its optimization can be achieved by strategically utilizing caching functionalities within an application's software framework and operating system. For instance, optimizing CPU cache usage can be achieved through performance tools like 'perf' on Linux and Intel's VTune tool, which can identify lines of code with a high cache miss rate.

2.3.5.1) Web Caching

Web caching involves caching data on both the web server and the client side (e.g., web browser, HTTP or SOAP service request initiator).

2.3.5.1.1) HTTP Cache Headers

Configuring the HTTP cache headers, such as Cache-Control and Expires [36], is an effective web caching technique. Cache-Control enables caching in the browser and intermediate proxies, while Expires sets the expiration date of a cached resource.

However, it's important for software developers to implement these headers correctly. In some cases, developers may encounter issues where changes made to a page's GUI, for example, are not reflected in the browser because it continues to load the previous cached version. Instead of disabling the cache using "Cache-Control:no-cache," a common but inefficient practice is to add a random fake HTTP attribute to the request (e.g., <img src='picture.jpg?123>) to prevent a cache hit. This approach wastes energy since it doesn't proactively disable caching when unnecessary.

2.3.5.1.2) Web Application-Level Caching

Web application-level caching involves programmatically caching data within the application's program execution through API calls or custom tag libraries imported into scripting languages like JSP. An example of application-level caching [30] is the Java Object Cache [29], which can be accessed through the Java Spring Caching Abstraction [31][32] in Spring or Spring Boot.

It is crucial to carefully consider and plan the implementation of web application-level caching to avoid redundant caching in the application's data flow, thereby optimizing energy efficiency.

2.3.5.1.3) Edge Side Includes

In certain scenarios, Edge Side Includes (ESI) [14], an XML-based markup language, can be used as an energy-efficient alternative to Client Side Includes (CSI) that relies on JavaScript and AJAX. ESI offers comparable load times while avoiding unnecessary requests. When CSI includes uncacheable AJAX response data, it requires multiple requests from the browser to the web server to fully load a page with dynamic fragments. In contrast, when a page is cached in the browser using ESI, it loads quickly, sends AJAX calls to the web server, and refreshes only the areas where response data is applied.

ESI enables the server to break down a dynamic web page into fragments, process them separately, and reassemble them before delivering the complete page to the browser. This method typically requires only one request to the web server, resulting in energy savings. ESI can be utilized on an edge server, such as a CDN edge node, and is also compatible with API calls that do not execute JavaScript.

2.3.5.2) Cloud Caching (cloud-specific)

Cloud caching involves using a managed web service to set up, operate, and scale a distributed web cache in the cloud. It provides the benefits of a high-performance in-memory cache with reduced administrative burden in managing a distributed cache. Additionally, users can configure the service to receive alarms if the cache becomes hot and access performance metrics through a user account page.

The energy efficiency of cloud caching depends on factors such as cache size, cache item sizes, and frequency of use. Notable examples of in-memory cloud caching solutions include AWS ElastiCache [33], Google Cloud Memorystore [34], and Microsoft Azure Cache for Redis [35]. These services offer energy-efficient caching capabilities tailored for cloud environments.

2.3.6) Optimize Search and Query Strategies

Optimizing search and query strategies can improve performance and save energy. The effectiveness of these tactics heavily relies on the type of database (relational or NoSQL) and the schema, which should be designed based on how the data will be utilized.

2.3.6.1) SQL Optimization

Optimizing SQL statements can reduce memory usage and disk access, and there are several techniques to achieve this. For instance, reducing table size, utilizing EXISTS() instead of COUNT(), implementing table indexes, using WHERE instead of HAVING, and adding EXPLAIN() at the beginning of a query to measure and optimize its performance during the optimization phase. SQL query optimization tools like the SolarWinds [49] Database Performance Analyzer can be employed to aid in this process.

2.3.6.2) NoSQL Optimization

Similar to SQL databases, there are multiple approaches to enhance the energy efficiency of NoSQL databases. One method involves storing records as variable-length delimited, which can significantly reduce their size compared to fixed-length record formats. Additionally, optimizing the layout of flat file data by logically and physically separating it and implementing sorting can narrow down the scope of records accessed during a search. This strategy improves energy efficiency, especially when the data layout is optimized for filtering methods such as regular expressions.

2.3.7) Compress Infrequently Accessed Data

Data compression is an effective way to save energy by reducing file storage and memory space. However, when transmitting compressed data over a computer network, especially over long distances, the positive impact on energy consumption can be even greater. It’s important to consider that compressing frequently accessed data may not be beneficial if the energy required for compression outweighs the energy saved during storage and movement over time. It’s worth noting that incompressible data, such as images with compressed data formats, should not be compressed.

2.3.7.1) Application Data Compression

Applications should implement real-time data compression when it is estimated to be advantageous. For example, in a Java EE Web application using the Spring Boot framework, Spring Boot can compress JSON data (using tools like Silz [16]) before transmitting it from the Web server to the browser. The compression type is recognized by the HTTP header Content-Encoding gzip, and the data is decompressed accordingly.

Another strategy in web development is to minimize JavaScript, CSS, XML, and JSON files. Minimization is performed during the build process, eliminating any additional energy or performance overhead during runtime. An example of a CSS and JavaScript compressor is YUI Compressor [45].

2.3.7.2) Cloud Database Compression (cloud-specific)

Certain cloud providers offer built-in compression capabilities. For instance, in the AWS RedShift [37] data warehouse service, database table column compression is automatically enabled by default unless explicitly configured otherwise. AWS provides documentation on Compression Encodings [38], which emphasizes the importance of evaluating table access frequency and identifying any incompressible data (such as graphics) before determining the appropriate configuration. It’s worth noting that compressing a table column may also result in a row offset.

2.3.8) Optimize Code

The four categories of techniques for application software energy efficiency are Computational Efficiency, Low-level or Intermediate Code Optimization, Parallelism, and Data and Communications Efficiency according to the Roskilde University’s Energy-Aware Software Development Methods and Tools research paper [6]. These are explained in the following sections (2.3.8.1 to 2.3.8.4).

2.3.8.1) Computational Efficiency

Achieving computational efficiency is crucial for optimizing energy consumption in applications, especially when considering single-threaded and multi-threaded scenarios. While the correlation between time and energy is clear for single-threaded applications, the distribution of functionality and resource-utilization density across threads in multi-threaded applications adds complexity to this relationship. However, the development of computationally efficient algorithms plays a significant role in improving the energy efficiency of both architectures.

2.3.8.1.1) Algorithm Design

Energy-efficient algorithm design relies on utilizing algorithms within their intended context and conditions, rather than applying a generic “one-size-fits-all” approach. A helpful perspective is to view energy-efficient algorithm design through the lens of Darwinian theory or the path of least resistance. Recent advancements in AI research, such as the work by scientists in Korea emulating the human brain [39], demonstrate the benefits of emulating nature’s optimization of resource utilization.

In their research, the scientists realized that the traditional “one-size-fits-all” approach to hardware design, where CPUs and memory utilize a fixed architecture, created a bottleneck. To address this constraint, they developed hardware that can dynamically alter its architecture, similar to the brain’s ability to change the connectivity structure of synapses. This innovative approach resulted in a 37% reduction in energy usage compared to current neural network implementations, without sacrificing accuracy. This breakthrough exemplifies how nature’s path of least resistance model can be applied to both architecture and algorithm design.

It is important to note that the fastest or smallest algorithms may not always be the most energy-efficient. Recursive algorithms, for instance, can be less energy-efficient than their iterative counterparts due to increased stack memory usage.

2.3.8.1.1.1) Energy-Efficient-Algorithm Design Process

Designing an energy-efficient software algorithm can involve following a holistic "path of least resistance" throughout the design process. This approach does not imply doing the minimal amount of work but rather consistently scoping efficiency to minimize the algorithm's energy footprint while maximizing performance.

Analogous to nature, which maintains efficiency through the path of least resistance, the energy-efficient algorithm design process can be visualized as shown in Figure 3 and described below:

1) Identify algorithm and variables:

Identify the algorithm's objective and its dependent and independent variables.

2) Optimize algorithm architecture:

Holistically minimize the compute time, memory requirements, and network utilization that are needed in order to reach this objective. Utilize knowledge of software energy efficiency best practices.

3) Assess data outlier probabilities:

Evaluate the impact of data outliers on the algorithm's energy efficiency. If the effect is significantly negative, re-evaluate the algorithm's architecture by returning to Step 2 to minimize any resulting overhead and inefficiency from additional code required to handle outliers.

4) Implement:

Once the algorithm design is complete, implement it with an understanding of the energy efficiency of programming languages and individual

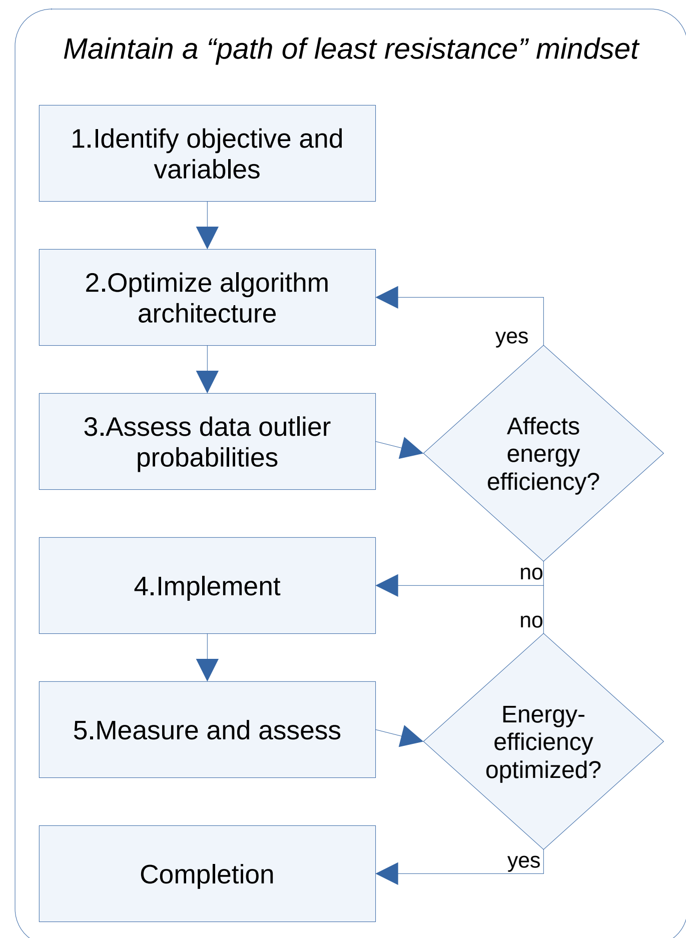


Figure 3: Energy-Efficient-Algorithm Design Process

language instructions (e.g., Java [1][57][58]) used in the application. Utilize energy-efficiency static analysis tools to aid in this process.

5) Measure and assess:

Employ dynamic analysis tools (section 3.1) to measure the energy efficiency of the software. If this is the initial iteration, use the measurement as a baseline for comparing successive iterations when seeking improvements. Repeat Step 5 until optimal energy efficiency is achieved.

2.3.8.1.2) Simplify Boolean-Logic Expressions

During software development, there are instances where long and complex boolean logic expressions are required. Simplifying these expressions can be done manually using techniques like Karnaugh maps or by utilizing online tools designed for this purpose, such as the Calculators.tech Boolean Algebra Calculator [40]. Simplification helps improve code readability, maintainability, and potentially reduces computational overhead, leading to enhanced energy efficiency.

2.3.8.2) Low-Level or Intermediate Code Optimization

Low-level software energy optimization techniques are typically performed within a compiler, a runtime engine (such as a JVM), or by selecting an energy-efficient programming language. Additionally, an application-specific power-scaling algorithm can be implemented in a middleware that acts as an interface between the runtime engine and the OS kernel's DVFS-based (dynamic voltage and frequency scaling) [7][8][9][10][11] power governor API. For more details on this implementation, refer to section 2.3.8.3.1 (DVFS) below.

Intermediate code optimizations focus on individual or small groups of code. It is ideal to start by using an energy-efficiency static analysis tool to identify and classify energy inefficiencies in the code. This tool can be likened to an application security vulnerability static analysis tool that automatically categorizes findings as Critical, High, Medium, or Low impact. If an energy static analysis tool is not available, manual code analysis becomes an alternative. In the case of Java, several research papers [1][57][58] delve into this topic.

2.3.8.2.1) Use the Most Optimal Computer Language

It is a common misconception that the fastest programming language is always the most energy-efficient. However, as Pereira et al. highlight [2], this is not always the case, as demonstrated in Table 1 below.

The speed and energy efficiency of a programming language can vary depending on the specific use case. Some languages offer universality and cross-platform capabilities, while others excel in specific tasks and environments. It's also important to consider how libraries and extensions are implemented,

as they can impact both speed and energy utilization. For example, the implementation of event handling can significantly influence energy efficiency. In most cases, interrupt-driven event handling proves to be more energy-efficient than using a polling mechanism.

Furthermore, employing multiple programming languages within an application can help optimize energy efficiency, speed, and memory usage, as each language can be leveraged for its strengths in different components or modules.

Table 1: Energy, speed, and memory-use of computer languages (Pereira et al. [2])

	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01

(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

(c)=compiled, (v)=virtual machine, (i)=interpreted

2.3.8.3) Parallelism

Multi-threaded applications that utilize multiple CPU cores generally exhibit higher energy efficiency compared to their single-threaded counterparts, thanks to their ability to effectively utilize computing resources. However, the energy efficiency of multi-threaded applications depends on how their functionality is distributed across process threads and the CPU cores that host them.

2.3.8.3.1) DVFS

In certain scenarios, multi-threaded applications can optimize their energy efficiency at runtime by utilizing a CPU's Dynamic Voltage Frequency Scaling (DVFS) [7][8][9][10][11] technology, which allows for controlling the frequency of its cores. By default, the OS kernel's driver automatically manages this technology. However, it is possible to override the default behavior by setting the CPU's power governor to "userspace" mode, enabling manual configuration. The OS's thread affinity runtime library can then be used to manually assign process threads with different speed requirements to cores operating at varying speeds.

It's worth noting that letting an application control these DVFS OS defaults can be particularly useful for applications with dedicated hardware, such as embedded IoT devices. In such cases, the CPU's energy profile can be tailored specifically for the application during selected phases of runtime.

In a multi-tenant environment, allowing a CPU to run multiple instances of the same application with identical energy profiles could potentially enhance energy efficiency compared to other scenarios. However, if the system architecture leads to conflicting energy profiles among processes without any synergistic effects, allowing the software application to directly control DVFS may defeat the purpose and potentially result in worse CPU energy efficiency compared to when the system governor manages it.

Additionally, it is important to note that due to architectural and hardware access restrictions, cloud-hosted applications cannot directly control their DVFS CPU registers. However, the underlying cloud orchestration implementer may choose to exercise control over DVFS settings.

2.3.8.3.1.1) Energy Profiling for DVFS

Assigning process threads of an application to specific CPU cores should be based on the application's energy profile. For instance, a CPU core can be designated to host an application thread that predominantly performs I/O operations and spends a significant amount of time in a wait state. Such a thread would be classified as low power in the application's energy-efficiency static analysis and can be assigned to a low-frequency CPU core with a similar classification during application initialization.

Initializing an application's energy parameters can involve running it in separate passes, with each subsequent pass refining the distribution of resources to DVFS-configured CPU cores. The settings derived from these passes can then be used for runtime execution. An example of this technique, as applied to Java applications, is presented in the research paper titled "Vincent: Green Hot Methods in the JVM" [9]. The authors' approach, implemented in their tool called VINCENT, achieves 14.9% energy savings compared to the built-in power management in Linux. The paper provides a high-level overview of the four execution passes as follows:

- **Phase 1: Hot Method Selection -** VINCENT obtains a list of hot methods.
- **Phase 2: Energy Profiling -** VINCENT profiles the energy consumption of hot methods under the default ONDEMAND governor (cpu), ranks their energy consumption, and reports a list of top energy-consuming methods as output.
- **Phase 3: Frequency Selection -** For each top energy-consuming method, VINCENT observes the energy consumption and execution time of the application at different CPU frequencies (configurations). It ranks the efficiency of different configurations based on energy metrics and selects the most efficient one for each method.
- **Phase 4: Energy Optimization -** VINCENT runs the application with each top energy-consuming method scaled to the CPU frequency determined in the Frequency Selection phase.

2.3.8.4) Data and Communications Efficiency

In a research paper by Roskilde University [6], the following is stated.

"Energy can be saved by minimizing data movement. This can be achieved by writing software that reduces data movement by using appropriate data structures, by understanding and exploiting the underlying system's memory hierarchy and by designing multi-threaded code that reduces the cost of communication among threads.

For example, the size of blocks read and written to memory and external storage can have a major impact on energy efficiency, while memory layout of compound data structures should match the intended usage in the algorithm, so that consecutively referenced data items are stored adjacently if possible. In multi-threaded code, consolidating all read-writes to or from disk to a single thread can

reduce disk contention and consequent disk-head thrashing. Furthermore, knowledge of the relative communication distances for inter-core communication can be used to place frequently communicating threads close to each other thus reducing communication energy costs."

3) Resource Monitoring

In the software energy-efficiency context, Resource Monitoring involves the monitoring and classification of workloads in order to optimize energy performance [41]. The following sections describe various tools for this purpose. The following sections contain tools for this purpose. For more tools, see a list of them posted by the Green Software Foundation's Innovation Working Group called "Awesome Green Software" [61], and in a research paper written by Acar et al [60].

3.1) Application Energy Monitoring

Currently, the ability to monitor application power is currently in its early stages of development. One reason for this is that in order for the associated tools to be accurate, they need to support many variations of operating systems, CPUs, and external dependencies such as services and libraries. Some projects are as follows:

- PowerAPI (powerapi.org) [23] - contains various tools for logging the energy consumption of the machine, program, and individual processes in devices with an Intel CPU.
- JoularJX [59] - Java-based agent for software power monitoring of Java methods and is compatible with Intel-based hardware. Available for Windows and Linux.

3.1.1) Cloud Energy Monitoring (cloud-specific)

Cloud providers have been able to provide robust energy tools because they exist in their own controlled and isolated infrastructure. Some of these tools are as follows.

Amazon Web Services (AWS):

Customer Carbon Footprint Tool [19] - Track, measure, review, and forecast the carbon emissions generated from your AWS usage. Available in the AWS Billing Console.

Microsoft Azure:

Microsoft Emissions Impact Dashboard [43] - Measures Microsoft Cloud emissions and carbon-saving potential. Available for Microsoft 365 and Microsoft Azure.

Google Cloud Platform (GCP):

Carbon Footprint Tool [42] - Provides users information showing the gross carbon emissions associated with their Google Cloud Platform usage. Available in the Cloud Console.

Open Source:

Cloud Carbon Footprint (GCF) tool [44] - calculates energy from a cloud provider's usage data and then applies the power usage effectiveness of the cloud provider's data centers and the carbon intensity of the data center region that is being used.

CodeCarbon tool [51] - Python package for tracking the carbon emissions produced by the cloud or personal computing resources used to execute the code. It also shows how the user can lessen emissions by optimizing their code or hosting their cloud infrastructure in geographical regions that use renewable energy sources.

Other:

Climatiq [50] - Using a REST api, convert cloud CPU and memory usage, storage, and networking traffic to CO2e estimates for Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure.

3.1.2) Performance Tools & Energy To Performance Correlation

There are many tools to help analyze software performance, and time to completion is a basis for a software task's energy use, but their use as an evaluator for energy optimization tactics should not be exclusive of tools that measure energy. This is because execution speed does not have a high correlation with energy efficiency in some cases. For example, when this correlation is low, an algorithm will not be optimized to efficiently handle a particular data flow rate and pattern (ie intermittent or continuous) but will be configured to run in a way that performance will not be affected. An example of this is a polling mechanism that runs at a poll rate that is high enough to have performance that equals an interrupt-driven equivalent implementation.

3.2) OS Power Monitoring

There are numerous power monitoring tools available for different operating systems. Some examples are as follows:

Linux:

powerTop [21], pTop, powerStat [22]

Windows, MacOS:

Intel PowerGadget [24], PowerCFG (comes with Windows OS)

3.3) Hardware Energy Monitoring

Although it cannot be used in a cloud environment, using a hardware energy-monitoring device in other situations can be useful under the following circumstances:

- The application's power utilization must make up the majority of its hardware capacity in order to be measurable when compared to the operating system's power utilization. For example, applications running on small embedded or IoT devices with a small or no RTOS (real-time operating system).
- If the application runs in a virtual machine or container, make sure sure that only one instance is running on the hardware platform that is being measured.
- Intermittent hardware functionality such as variable cpu fans and battery charging must not exist or should be disabled if possible.
- OS cron jobs and daemons that will impact the energy measurements should be disabled if possible.
- If only a wattage measurement is available on the energy-monitoring device, then measuring the energy (E=Power*Time) use will require the manual collection of measurements over a period of time in order to calculate it.
- Manual sampling may produce an inaccurate power measurement because assumptions will need to be made for power-use over periods of time that exceed the sampling period. Full automation of data sampling and energy calculations over long periods of time and under a load that reflects common, long-term and consistent scenarios is ideal.
- The device's energy reading should be WattHours (Wh) and needs to be accurate to at least two decimal places if sampling measurements for less than 10 minutes, and this duration will probably be the case when manually writing down measurements in the case where the hardware device has no logging support. A kWh reading is not useful for this purpose because it would require at least five decimal places of accuracy to be useful (offering >= 2 decimal places of Wh accuracy), and devices that only offer this energy reading usually only go to two decimal places.
- The energy reading should have a reset feature so that it can be cleared before starting an energy measurement.

3.3.1) Measuring Application Energy using a Hardware Power Monitor

To get a rough estimate of the application's power usage, install a baseline OS image, remove its intermittent processes where possible, and then obtain the device's energy utilization with the power

monitoring device. At this point, install the application, get the energy again, and then subtract the baseline OS energy measurements from it.

Ideally, the power measurement device should contain an energy measurement in WattHours (Wh) and have an accuracy of at least two decimal places. If it shows only the current wattage, then calculate the energy (Energy=Power*Time) manually by first deciding how long and at what frequency to take measurements. For example, if taking one measurement every second for one minute, then calculate the average of the samples over that period of time and if the result is 10 watts, then the energy used during this time is 10 watt-minutes which is 0.17 Wh (10/60).

3.3.2) Hardware Power Monitoring Devices

The following hardware devices are reasonably priced and offer WattHour (Wh) energy measurements and data collection capabilities. Finding other devices is a work in progress.

Watts Up? Pro [56]: (*note: This product is discontinued by the manufacturer*).

HOBO UX120-018 Plug Load Data Logger. For 120v/50hz. (*note: 240V/220V/50hz available?*)

Conclusion

The increase in energy prices and the worldwide plan to enforce policies for low carbon emissions will likely continue to accelerate while forcing companies and individuals to adapt. As the ICT industry heads into this new era, its software energy-efficiency concerns will continue to accelerate.

Providing the groundwork of software energy-efficiency architecture and implementation tactics, processes, and tools is fundamental to providing a solution to this problem. This whitepaper helps to partially fulfill that requirement through its use as a software energy-efficiency tactic reference guide during the initial stages of this process.

Learn How Improving Software Energy-Efficiency Can Help You

If you are considering the value in making your software more energy-efficient, take advantage of Agile7's personalized approach and commitment to building strong customer relationships. You can count on us to help you assess your software's energy footprint, determine how it can be optimized, and implement the changes necessary to improve its energy efficiency while increasing or maintaining its performance, all while matching your schedule, cost requirements, personnel and policies. We can provide training, accompany sellers on projects, directly support the end user, and all while consistently prioritizing our partner relationships.

Please contact us at **info@agile7.com**.

About Agile7

Agile7 supplies software development services for software energy-efficiency and application vulnerability remediation. Learn more at **www.agile7.com**.

Document Revision History

Author	Version	Date	Description
Max Meinhardt	1.0	July 10, 2022	Initial version
Max Meinhardt	1.1	October 23, 2024	Changed logo and added Document Version on title page, added Version column to Document Revision History table, and corrected grammar throughout the document.

Author Biography

About Max Meinhardt

Max is the founder of Agile7 and he holds over three decades of industry experience in software engineering and systems deployment leading the architecture and implementation of enterprise Web applications and carrier-class networking and telecommunications equipment firmware. He holds a BS in Computer Engineering Technology from Rochester Institute of Technology and an MBA from Thunderbird School of Global Management.

References

1 Kumar, Mohit. "Improving Energy Consumption Of Java Programs" (2019). Wayne State University Dissertations. 2325.

2 Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency across Programming Languages. In Proceedings of SLE'17, Vancouver, BC, Canada, October 23–24, 2017, 12 pages. https://doi.org/10.1145/3136014.3136031. accessed on 2022-05-07.

3 Dave McCarthy. 2020. White Paper. AWS at the Edge: A Cloud Without Boundaries. Sponsored by: Amazon Web Services. IDC.

4 Amazon Web Services. AWS for the Edge. https://aws.amazon.com/edge/services, accessed on 2022-0625.

6 Roskilde University, University of Bristol, IMDEA Software Institute, XMOS Limited. March 1, 2016. ENTRA 318337 - Whole-Systems ENergy TRAnsparency - Energy-Aware Software Development Methods and Tools. In beneficiary of IMDEA Software Institute. http://entraproject.ruc.dk/wp-content/uploads/2016/03/deliv_1.2.pdf. accessed on 2022-06-27.

7 Rafael J. Wysocki, 2017. CPU Performance Scaling. Intel Corporation. https://www.kernel.org/doc/html/v4.12/admin-guide/pm/cpufreq.html. accessed on 2022-06-22.

8 Rafael J. Wysocki, 2017. intel_pstate CPU Performance Scaling Driver. https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html. accessed on 2022-06-25

9 Kenan Liu, Khaled Mahmoud, Joonhwan Yoo, Yu David Liu. 2022. Vincent: Green Hot Methods in the JVM, 29 pages.

10 Kenan Liu, Gustavo Pinto, Yu David Liu. 2015. Data-Oriented Characterization of Application-Level Energy Optimization. 12 pages.

11 Timur Babakol, Anthony Canino, Khaled Mahmoud, Rachit Saxena, Yu David Liu. 2020. Calm Energy Accounting for Multithreaded Java Applications. 8 pages.

12 Gartner Glossary - Information Technology. https://www.gartner.com/en/information-technology/glossary/cloud-services-brokerage-csb, accessed on 2022-06-25.

13 L. Vailshery. Number of IoT connected devices worldwide 2019-2030. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/. accessed on 2022-06-25.

14 M. Tsimelzon, B. Weihl, J. Chung, D. Frantz, J. Basso, C. Newton, M. Hale, L. Jacobs, C O'Connell. ESI Language Specification 1.0. W3C Note 04 August 2001. https://www.w3.org/TR/esi-lang/. accessed on 2022-06-25.

16 Karsten Silz. Reducing JSON Data Size. June 24, 2022. https://www.baeldung.com/json-reduce-data-size.

accessed on 2022-06-25.

17 Amazon Web Services. How the AMS Resource Scheduler works. https://docs.aws.amazon.com/managedservices/latest/userguide/resource-scheduler-how-works.html. accessed on 2022-06-25.

18 Jason Deszkewicz, "Difference Between Static and Dynamic Web Pages", https://www.academia.edu/23686425/Difference_Between_Static_and_Dynamic_Web_Pages. accessed on 2022-06-25.

19 Amazon Web Services. Customer Carbon Footprint Tool. https://aws.amazon.com/aws-cost-management/aws-customer-carbon-footprint-tool. accessed on 2022-06-25.

20 AppDynamics. https://appdynamics.com. accessed on 2022-06-25.

21 Arjan Van de Ven, fenrus/powertop. Github repository. https://01.org/powertop/. accessed on 2022-06-25.

22 Colin King. Powerstat computer power measuring tool. Last updated 2022-17-09. https://snapcraft.io/powerstat. accessed on 2022-06-25.

23 Aurelien Bourdon. PowerAPI. Spirals Research Group (University of Lille and Inria). https://abourdon.github.io/powerapi-akka/. accessed on 2022-06-25.

24 Intel. Intel Power Gadget. https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html. accessed on 2022-07-05.

25 Greenspector. "Mobile Efficiency Index. Measure the efficiency of your website!" http://mobile-efficiency-index.com/en/. accessed on 2022-05-07.

26 Tom Nardi. A Complete Raspberry Pi Power Monitoring System. Hackaday. https://hackaday.com/2020/07/24/a-complete-raspberry-pi-power-monitoring-system/. Published on 2020-05-07. website accessed on 2022-05-07.

28 Shelly. Shelly Plug. https://github.com/LLNL/msr-safe, https://shelly.cloud/products/shelly-plug-smart-home-automation-device/. accessed on 2022-05-07.

29 Oracle. Working with Java Object Cache. Oracle. Oracle9/AS Containers for J2EE Services Guide. Release 2 (9.0.3) https://docs.oracle.com/cd/B10002_01/generic.903/a97690/objcache.htm. accessed on 2022-05-07.

30 Jhonny Mertz, Ingrid Nunes. Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches. ACM Computing Surveys, Volume 50, Issue 6, Nov 2018, Article No.:98. 34 pages. https://dl.acm.org/doi/10.1145/3145813. accessed on 2022-05-07.

31 Eugen Paraschiv. A Guide To Caching in Spring. Baeldung. https://www.baeldung.com/spring-cache-tutorial. accessed on 2022-05-07.

32 Spring. Cache Abstraction. https://docs.spring.io/spring-framework/docs/5.0.0.M1/spring-framework-

reference/html/cache.html. accessed on 2022-05-07.

33 Amazon Web Services. "Amazon ElastiCache. Unlock microsecond latency and scale with in-memory caching." https://aws.amazon.com/elasticache/. accessed on 2022-05-07.

34 Google Cloud. Memorystore. https://cloud.google.com/memorystore/. accessed on 2022-05-07.

35 Microsoft Azure. "Azure Cache for Redis. Distributed, in-memory, scalable solution providing super-fast data access." https://azure.microsoft.com/en-us/services/cache/#overview. accessed on 2022-05-07.

36 Heroku Dev Center. Increasing Application Performance with HTTP Cache Headers. https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers. Last updated on 2022-09-03. accessed on 2022-05-07.

37 Amazon Web Services. Compression encodings. Amazon Redshift - Database Developer Guide. https://docs.aws.amazon.com/redshift/latest/dg/c_Compression_encodings.html. accessed on 2022-05-07.

38 Amazon Web Services. Amazon Redshift Engineering's Advanced Table Design Playbook: Compression Encodings. https://aws.amazon.com/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compression-encodings/. accessed on 2022-06-26.

39 ScienceDaily. "Energy-efficient AI hardware technology via brain-inspired stashing system?" Published on 2022-17-05. Science Daily sourced from the Korea Advanced Institute of Science and Technology (KAIST). https://www.sciencedaily.com/releases/2022/05/220517210435.htm. accessed on 2022-05-07.

40 Calculators.tech. Boolean Algebra Calculator. https://www.calculators.tech/boolean-algebra-calculator. accessed on 2022-06-26.

41 Sophie Vos, Patricia Lago, Roberto Verdecchia, Ilja Heitlager. Architectural Tactics to Optimize Software for Energy Efficiency in the Public Cloud. 2022, 11 pages.

42 Google Cloud. Google Cloud Carbon Footprint Tool. https://cloud.google.com/carbon-footprint. accessed on: 2022-06-21.

43 Microsoft. Emissions Impact Dashboard. https://www.microsoft.com/en-us/sustainability/emissions-impact-dashboard. accessed on: 2022-06-21.

44 Cloud Carbon Footprint. "Cloud Carbon Footprint - Free and Open Source - Cloud Carbon Emissions Measurement and Analysis Tool", https://www.cloudcarbonfootprint.org. accessed on 2022-06-21.

45 YUI. YUI Compressor. https://yui.github.io/yuicompressor/

46 C. Paradis, R. Kazman, and D. A. Tamburri, “Architectural tactics for energy efficiency: Review of the literature and research roadmap,” in Proceedings of the 54th Hawaii International Conference on System Sciences. Hawaii: ScholarSpace, 2021, pp. 7197–7206. https://hdl.handle.net/10125/71488. accessed on 2022-05-07.

47 Amazon Web Services. "AWS Snowball. Accelerate moving offline data or remote storage to the cloud."

https://aws.amazon.com/snowball/, accessed on 2022-06-23.

48 Amazon Web Services. Scheduled scaling for Amazon EC2 Auto Scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-scheduled-scaling.html. accessed on 2022-06-24.

49 SolarWinds. Database Performance Analyzer. https://www.solarwinds.com/database-performance-analyzer. accessed on 2022-06-25

50 Climatiq. Cloud Computing Carbon Emissions. https://www.climatiq.io/cloud-computing-carbon-emissions. accessed on 2022-06-27.

51 CodeCarbon. https://codecarbon.io/. accessed on 2022-06-27.

55 Stefanos Georgiou, Stamatia Rizou, Diomidis Spinellis, 2019. Software Development Life Cycle for Energy Efficiency: Techniques and Tools. ACM Comput. Surv. 1, 1, Article 1 (January 2019), 35 pages. DOI: 10.1145/3337773

56 J. M. Hirst, J. R. Miller, B. A. Kaplan, and D. D. Reed, "Watts up? pro ac power meter for automated energy recording," ed: Springer, 2013.

57 Mohit Kumar, Youhuizi Li, Weisong Shi. "Energy Consumption in Java: An Early Experience" (2017). 2017 IGSC Track on Contemporary Issues on Sustainable Computing.

58 Gustavo Pinto, Kenan Liu, Fernando Castor, Yu David Liu. "A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections" (2016).

59 Adel Noureddine. JoularJX. GitLab. https://gitlab.com/joular/joularjx. accessed on 2022-07-07.

60 Hayri Acar. Software development methodology in a Green IT environment. Other [cs.OH]. Université de Lyon, 2017. English. ffNNT : 2017LYSE1256ff. fftel-01724069f. https://tel.archives-ouvertes.fr/tel-01724069/document. p37-50. accessed on 2022-07-07.

61 Green Software Foundation. Awesome Green Software - Green Software - Research, tools, code, libraries, and training for building applications that emit less carbon into our atmosphere. https://github.com/Green-Software-Foundation/awesome-green-software. accessed on 2022-07-07.